# Feature Location Benchmark for Software Families using Eclipse Community Releases

Jabier Martinez[1,2], Tewfik Ziadi[2], Mike Papadakis[1], Tegawendé F. Bissyandé[1], Jacques Klein[1], and Yves le Traon[1]

[1] SnT, University of Luxembourg
Luxembourg, Luxembourg
name.surname@uni.lu
[2] LiP6, Sorbonne Universités, UPMC Univ Paris 06
Paris, France
tewfik.ziadi@lip6.fr

**Abstract.** It is common belief that high impact research in software reuse requires assessment in realistic, non-trivial, comparable, and reproducible settings. However, real software artefacts and common representations are usually unavailable. Also, establishing a representative ground truth is a challenging and debatable subject. Feature location in the context of software families is a research field that is becoming more mature with a high proliferation of techniques. We present EFLBench, a benchmark and a framework to provide a common ground for this field. EFLBench leverages the efforts made by the Eclipse Community which provides real feature-based family artefacts and their implementations. Eclipse is an active and non-trivial project and thus, it establishes an unbiased ground truth. EFLBench is publicly available and supports all tasks for feature location techniques integration, benchmark construction and benchmark usage. We demonstrate its usage and its simplicity and reproducibility by comparing four techniques.

**Keywords:** Feature location, software product lines, benchmark, static analysis, information retrieval

## 1 Introduction

Software reuse is often performed by industrial practitioners mainly to boost productivity. One such case is the *copy-paste-modify* which is performed when creating product variants for supporting different customer needs [9]. This practice may increase the productivity in a short term period but in the long run it becomes problematic due to the complex maintenance and further evolution activities of the variants [4]. To deal with these issues, Software Product Line (SPL) engineering has developed mature techniques that can support commonality and variability management of a whole product family and the derivation of tailored products by combining reusable assets [4].

Despite the advantages provided by SPLs, their adoption still remains a major challenge because of organizational and technical issues. To deal with it,

software reuse community proposed the so-called *extractive* or *bottom-up* approaches. Among the various bottom-up processes, in this paper we focus on feature location. A feature is defined as a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems [15]. As pointed out in the surveys of Rubin *et al.* and Assunção *et al.* [5, 24], feature location is an important and challenging problem of these bottom-up processes towards systematic reuse. Many approaches have been proposed and there is a progression in the number of research work conducted every year [5]. Thus, it can be stated that there is an increasing interest on the topic by the research community [19].

Comparing, evaluating and experimenting with feature location techniques is challenging due to the following reasons:

- Most of the research prototypes are either unavailable or hard to configure.
- Performance comparison requires common settings and environments.
- Most of the tools are strongly dependent on specific artefact types that they were designed for, e.g., programming language, design models, etc.
- Effectiveness of the techniques can vary according to different implementation element types, e.g., Abstract Syntax Tree (AST) nodes, software components, etc, that are to be located.

Common case study subjects and frameworks are in need to foster the research activity [30]. In this direction, we identified a set of requirements for such frameworks in feature location:

**A standard case study subject.** Subjects that are real, non-trivial and easy to use are mandatory. This includes: 1) A list of existing features; 2) For each feature, a group of elements that implements it. 3) A set of real product variants accompanied by the information of which features are included.

**A benchmarking framework.** In addition to the standard subjects, a full implementation that allows a common, quick and intensive evaluation is needed. This includes 1) available implementation with a common abstraction for the product variants to be considered by the case studies, i.e., as unified structured elements; 2) easy and extensible mechanisms to integrate feature location techniques to support the experimentation and 3) sets of predefined evaluation metrics to draw comparable results.

This paper proposes a framework, called **Eclipse Feature Location Benchmark (EFLBench)**, that fulfils the requirements identified above. We propose a standard and a realistic case study for feature location and an integrated benchmark using the packages of Eclipse releases, their features and their associated plugins. We also propose a full implementation to support benchmarking within Bottom-Up Technologies for Reuse (BUT4Reuse) [21] that is an open-source, generic and extensible framework for bottom-up approaches which allows a quick integration of feature location techniques.

The rest of the paper is structured as follows: Section 2 provides background information about feature location techniques and the Eclipse project. In Section 3 we present Eclipse as a case study subject and then in Section 4 we present the EFLBench framework. Section 5 presents different feature location techniques and the results of EFLBench usage. Section 6 presents related work and Section 7 concludes and presents future work.

## 2 Background

In order to provide a better understanding for the following sections of this paper, we provide details about feature location and about the Eclipse project.

### 2.1 Feature Location

Bottom-up approaches for SPL adoption are mainly composed of the following processes: *Feature identification*, *feature location* and *re-engineering* [21]. While feature identification is the process that takes as input a set of product variants and analyses them to discover and identify features, the feature location is the process of mapping features to their concrete implementation in the product variants. Therefore, compared to the feature identification process, the assumption in feature location is that the features are known upfront. Feature location processes in software families also use to assume that feature presence or absence in the product variants is known upfront. However, what is unknown is where exactly they are implemented inside the variants. Finally, feature re-engineering is the process that includes a transformation phase where the artefact variants are refactored to conform to an SPL approach. This includes extracting, for each feature, reusable assets from the artefact variants.

As already mentioned, the objective of feature location approaches is to map features to their concrete implementation parts inside the product variants. However, depending on the nature of the variants, this can concern code fragments in the case of source code [2, 11, 23, 34], model fragments in the context of models [12, 20] or software components in software architectures [1, 14]. Therefore, existing techniques are composed of the following two phases: 1) *Abstraction*, where the different product variants are abstracted and represented as implementation elements; 2) *Location*, where algorithms analyse and compare the different product variants to create *groups of implementation elements*. These groups are to be associated with the sought features. Despite these two phases, feature location techniques differ in the following three aspects:

- **The way the product variants are abstracted and represented.** Indeed, each approach uses a specific formalism to represent product variants. For example AST nodes for source code [11], Atomic-Model-Element to represent model variants [20] or plugins in software architectures [1]. In addition, the granularity of the sought implementation elements may vary from coarse to fine [16]. Some use fine granularity using AST nodes that cover all source code statements while others use purposely a little bit bigger granularity using object-oriented building elements [2] like Salman *et al.* that only consider classes [25].
- **The proposed algorithms.** Each approach proposes its own algorithm to analyse product variants and identify the groups of elements that are related to features. For instance, Fischer *et al.* [11] used a static analysis algorithm. Other approaches use techniques from the field of Information Retrieval (IR). Xue *et al.* [33] and Salman *et al.* [26] proposed the use of Formal Concept Analysis (FCA) to group implementation elements and then, in a second step, the IR technique Latent Semantic Indexing (LSI) to map between

these groups and the features. Salman *et al.* used Hierarchical Clustering to perform this second step [25].

– **The used case studies to evaluate and experiment the proposed technique.** The evaluation of each technique is often performed using its own case study and with its own evaluation measures.

### 2.2 The Eclipse Project

The Eclipse community, with the support of the Eclipse Foundation, provides integrated development environments (IDE) targeting different developer profiles. The project IDEs cover the development needs of *Java*, *C/C++*, *JavaEE*, *Scout*, *Domain Specific Languages*, *Modeling*, *Rich Client Platforms*, *Remote Applications Platforms*, *Testing*, *Reporting*, *Parallel Applications* or for *Mobile Applications*. Following Eclipse terminology, each of the customized Eclipse IDEs is called an Eclipse **package**.

As the project evolves over time, new packages appear and some other ones disappear depending on the interest and needs of the community. For instance, in 2011 there were 12 packages while the next year 13 packages were available with the addition of one targeted to *Automotive Software* developers.

Continuing with Eclipse terminology, a *simultaneous release* (**release** hereafter) is a set of packages which are public under the supervision of the Eclipse Foundation. Every year, there is one main release, in June, which is followed by two service releases for maintenance purposes: SR1 and SR2 usually around September and February. For each release, the platform version changes and traditionally celestial bodies are used to name the releases, for example Luna for version 4.4 and Mars for version 4.5.

The packages present variation depending on the included and not-included **features**. For example, Eclipse package for Testers is the only one that includes the Jubula Functional Testing features. On the contrary, other features like the Java Development tools are shared by most of the packages. There are also common features for all the packages, like the Equinox features that implement the core functionality of the Eclipse architecture. The online documentation of each release provides a high-level information of the features that each package provides [1].

It is important to mention that in this work we are not interested in the variation among the releases (version 4.4, 4.5 and so on), known as *variation in time*, because this is related to software maintenance and evolution. We focus on the variation of the different packages of a given release, known as *variation in space*, which is expressed in terms of included and not-included features. Each package is different in order to support the needs of the targeted developer profile by including only the appropriate features.

Eclipse is feature oriented and based on **plugins**. Each feature consists of a set of plugins that are the actual implementation of the feature. Table 1 shows an example of feature with four plugins as implementation elements that, if included

[1] `https://eclipse.org/downloads/compare.php?release=kepler`

in an Eclipse package, adds support for a versioning system based on CVS. At technical level, the actual features of a package can be found within a folder called *features*. This folder contains meta-information regarding the installed features including the list of plugins associated to each of the features. Each feature has an id, a name and a description as written by the feature providers of the Eclipse community. A plugin has an id and a name written by the plugin providers but it does not have a description.

Table 2 presents data regarding the evolution of the Eclipse releases over the years. In particular, it presents the total number of packages, features and plugins per release. To illustrate the distribution of packages and features in the project, Figure 1 depicts a matrix of the different Eclipse Kepler SR2 packages. In this figure, a black box denotes the presence of a feature (horizontal axis) in a package (vertical axis). We observe that some features are present in all the packages while others are specific to only few, one or two, packages. The

**Table 1.** Eclipse feature example

| **Feature** | |
|---|---|
| *id:* org.eclipse.cvs | |
| *name:* Eclipse CVS Client | |
| *description:* Eclipse CVS Client (binary runtime and user documentation). | |
| **Plugin id** | **Plugin name** |
| org.eclipse.cvs | Eclipse CVS Client |
| org.eclipse.team.cvs.core | CVS Team Provider Core |
| org.eclipse.team.cvs.ssh2 | CVS SSH2 |
| org.eclipse.team.cvs.ui | CVS Team Provider UI |

**Table 2.** Eclipse releases and their number of packages, features and plugins

| **Year** | **Release** | **Packages** | **Features** | **Plugins** |
|---|---|---|---|---|
| 2008 | Europa Winter | 4 | 91 | 484 |
| 2009 | Ganymede SR2 | 7 | 291 | 1,290 |
| 2010 | Galileo SR2 | 10 | 341 | 1,658 |
| 2011 | Helios SR2 | 12 | 320 | 1,508 |
| 2012 | Indigo SR2 | 12 | 347 | 1,725 |
| 2013 | Juno SR2 | 13 | 406 | 2,008 |
| 2014 | Kepler SR2 | 12 | 437 | 2,043 |
| 2015 | Luna SR2 | 13 | 533 | 2,377 |

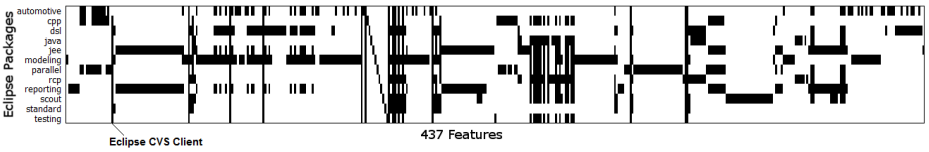


**Fig. 1.** Eclipse Kepler SR2 packages and a mapping to their 437 features

437 features are alphabetically ordered by their id and, for instance, the feature *Eclipse CVS Client*, tagged in Figure 1, is present in all of the packages except in the *Automotive Software* package.

Features, as in most of the feature oriented systems, have dependencies among them. *Includes* is the Eclipse terminology to define subfeatures and *Requires* means that there is a functional dependency between the features. Figure 2 shows the dependencies between all the features of Eclipse Kepler SR2. We tagged in Figure 2 some features and subfeatures of the Eclipse Modeling Framework. Functional dependencies are mainly motivated by the existence of dependencies between the plugins of one feature with the plugins of other features.

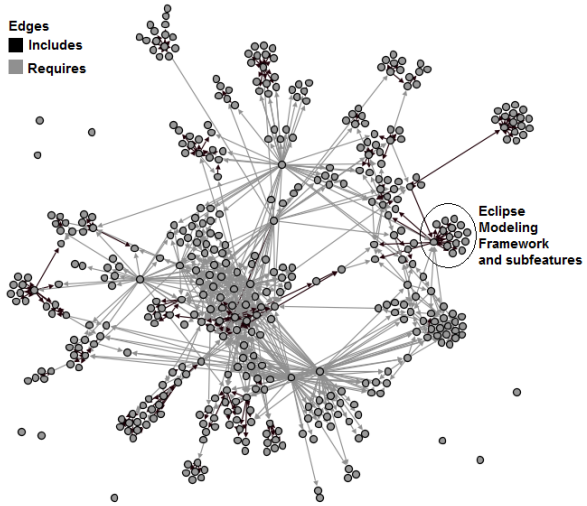


**Fig. 2.** Features of Eclipse Kepler SR2 and their dependencies

Both Feature and Plugin dependencies are explicitly declared in their metadata. Figure 3 shows a very small excerpt of the dependency connections of the 2043 plugins of Eclipse Kepler SR2. Concretely, this excerpt shows the dependencies of the four CVS plugins presented in Table 1.

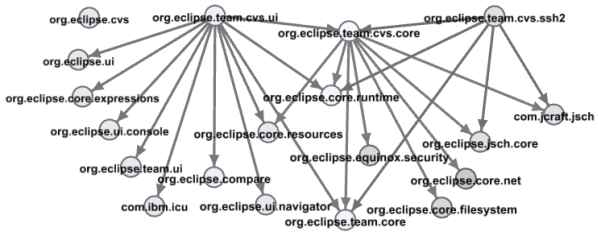


**Fig. 3.** Excerpt of plugin dependencies focusing on the dependencies of CVS plugins

## 3 Eclipse as a standard case study subject

Eclipse packages are an interesting candidate as a standard case study for a feature location benchmark. First, as mentioned in Section 1, it fulfils the requirement of providing the needed data to be used as ground truth. This ground truth can be extracted from features meta-information. Apart from this, Eclipse packages present other characteristics that make this case study interesting and challenging. This section aims to discuss these characteristics.

The relation between the number of available packages in the different Eclipse releases and the number of different features is not balanced. In fact, the number of available product variants has been shown to be an important factor for feature location techniques [11]. The limited number of packages and the big amount of features make the Eclipse case study challenging. The granularity of the implementation elements (plugins) is very coarse if we compare it with source code AST nodes, however, the number of plugins is still reasonably high. In Eclipse Kepler SR2, the total of plugins with different ids is 2043 with an average of 609 plugins per Eclipse package and a standard deviation of 192.

Eclipse feature and plugin providers have created their own natural language corpora. The feature and plugin names (and the description in the case of the features) can be categorized as meaningful names [24] enabling the use of several IR techniques. Also, the dependencies between features and dependencies between implementation elements have been used in feature location techniques. For example, in source code, program dependence analysis has been used by exploiting program dependence graphs [7]. Acher *et al.* [1] also leveraged architecture and plugin dependencies. As presented in previous section, Eclipse also has dependencies between features and dependencies between plugins enabling their exploitation during feature location.

There are properties that can be considered as "noise" that are common in real scenarios. Some of them can be considered as non-conformities in feature specification [31]. A case study without "noise" should be considered as a very optimistic case study. In Eclipse Kepler SR2, 8 plugins do not have a name and different plugins from the same feature have also exactly the same names. There are also 177 plugins which are associated to more than one feature. Thereby the features' plugin sets are not completely disjoint. These plugins are mostly related to libraries for common functionalities that they were not included as required plugins but as a part of the feature itself. In addition, 40 plugins present in some of the variants are not declared in any feature. Also, in few cases, feature versions are different among packages of the same release.

Apart from the official releases, software engineering practitioners have created their own Eclipse packages. Therefore, also researchers can use their own packages or create variants with specific characteristics. Interest of analysing plugin-based or component-based software system families to exploit their feature variability has been shown in previous works [1, 14, 29]. For instance, experiences in an industrial case study were reported by Grünbacher *et al.* [14] where they performed manual feature location in Eclipse packages to extract an SPL that involved more than 20 package customizations per year.

## 4 Eclipse Feature Location Benchmarking Framework

EFLBench is aimed to be used with any set of Eclipse packages. The benchmark can be created from any set of Eclipse packages that can have additional features which are not part of any official release. However, to set a common scenario for research we recommend and propose the use of Eclipse Community releases.

Figure 4, in the top part, illustrates the mechanism for constructing the benchmark taking as input the Eclipse packages and automatically producing two outputs, a) a Feature list with information about each feature name, description and the list of packages where it was present, and b) a ground truth with the mapping between the features and the implementation elements which are the plugins.

Once the benchmark is constructed, the bottom part of Figure 4 illustrates how it can be used through BUT4Reuse [21] where feature location techniques can be integrated. The Eclipse adapter, which is responsible for the variants abstraction phase, will be followed by the launch of the targeted feature location techniques. This process takes as input the Eclipse packages (excluding the *features* folder) and the feature list. The feature location technique produces a mapping between features and plugins that can be evaluated against the ground truth obtained in the benchmark construction phase.

The following subsections provide more details on the two phases.

### 4.1 Benchmark construction

We implemented an automatic extractor of features information. The implementation elements of a feature are those plugins that are directly associated to this feature. From the 437 features of the Eclipse Kepler SR2, each one has an average of 5.23 plugins associated with. The standard deviation is 9.67. There is one outlier with 119 plugins which is the feature *BIRT Framework* present in the Reporting package. From the 437 features, there are 19 features that do not contain any plugins, so they are considered *abstract* features which are created just for grouping other features. For example, the feature *UML2 Extender SDK*

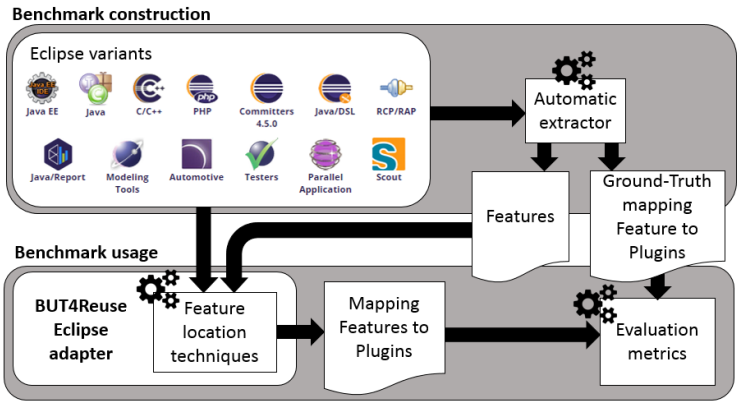


**Fig. 4.** EFLBench: Eclipse package variants as benchmark for feature location

(Software Development Kit) groups *UML2 End-User Features*, *Source for UML2 End-User Features*, *UML2 Documentation* and *UML2 Examples*.

Reproducibility is becoming quite easy by using benchmarks and common frameworks that launch and compare different techniques [30]. This practice, allows a valid performance comparison with all the implemented and future techniques. BUT4Reuse public repository includes EFLBench and its automatic extractor.

### 4.2 Benchmark usage

During the product abstraction phase, the implemented Eclipse adapter decomposes any Eclipse installation in a set of plugins by visiting and analysing the Eclipse installation file structure. The plugin elements contain information about their id, name as well as their dependency to other plugin elements.

At technical level, BUT4Reuse provides an extension point and interface to easily include feature location techniques [2]. After feature location, it calculates the *precision* and *recall* for each feature location technique which are classical evaluation metrics in IR studies (e.g., [25]). We explain precision and recall, two metrics that complements each other, in the context of EFLBench. A feature location technique assigns a set of plugins to each feature. In this set, there can be some plugins that are actually correct according to the ground-truth, those are *true positives* (TP). TPs are also referred to as *hit*. On the set of plugins retrieved by the feature location technique for each feature, there can be other plugins that do not belong to the feature, those are *false positives* (FP) which are also referred to as *false alarms*. Precision is the percentage of correctly retrieved plugins from the total of retrieved plugins by the feature location technique. A precision of 100% means that the ground truth of the plugins assigned to a feature and the retrieved set from the feature location technique are the same and no "extra" plugins were included. The formula of precision is as follows:

$$precision = \frac{TP}{TP + FP} = \frac{plugins\,hit}{plugins\,hit\ +\ plugins\,false\,alarm}$$

According to the ground truth there can be some plugins that are not included in the retrieved set, meaning that they are *miss*. Those plugins are *false negatives* (FN). Recall is the percentage of correctly retrieved plugins from the set of the ground-truth. A recall of 100% means that all the plugins of the ground-truth were assigned to the feature. The formula of recall is as follows:

$$recall = \frac{TP}{TP + FN} = \frac{plugins\,hit}{plugins\,hit\ +\ plugins\,miss}$$

Precision and recall are calculated for each feature. In order to have a global result of the precision and recall we use the mean of all the features. Finally, BUT4Reuse reports the *time* spent for the feature location technique. With this information, the time performance of different techniques can be compared.

[2] Instructions to integrate feature location techniques in BUT4Reuse: `https://github.com/but4reuse/but4reuse/wiki/ExtensionsManual`

## 5 Example of EFLBench usage

This section aims at presenting the possibilities of EFLBench by benchmarking four feature location techniques. The four techniques are using Formal Concept Analysis (FCA) and three of them are using natural language processing (NLP). Before enumerating the four techniques, we briefly present FCA and the used NLP algorithms.

### 5.1 Background algorithms

For the four techniques we used FCA [13] for the identification of an initial set of groups of implementation elements. We will refer to the identification of this initial set as block identification [21]. FCA groups elements that share common attributes. A detailed explanation about FCA formalism in the same context of block identification can be found in Al-Msie'deen *et al.* [2] and Shatnawi *et al.* [29]. FCA uses a *formal context* as input. In our case, the entities of the formal context are the Eclipse packages and the attributes (binary) are the presence or not of each of the plugins. With this input, FCA discovers a set of *concepts*. The concepts which contain at least one plugin (non empty concept intent in FCA terminology) is considered as a block. For example, in Eclipse Kepler SR2, FCA-based block identification identifies 60 blocks with an average of 34 plugins per block and a standard deviation of 54 plugins. In Eclipse Europa Winter, with only 4 packages, only 6 blocks are identified with an average of 80 plugins each and a standard deviation of 81. Given the low number of Eclipse packages, FCA identifies a low number of blocks. The number of blocks is specially low if we compare it with the actual number of features that we aim to locate. For example 60 blocks in Kepler SR2 against its 437 features. The higher the number of Eclipse packages, the most likely FCA will be able to distinguish different blocks. At technical level, we implemented FCA for block identification using ERCA [10].

In the approaches where we use IR techniques, we did not make use of the feature or plugin ids. In order to extract the meaningful words from both features (name and description) and elements (plugin names), we used two well established techniques in the IR field:

- Parts-of-speech tags remover: These techniques analyse and tag words depending on their role in the text. The objective is to filter and keep only the potentially relevant words. For example, conjunctions (f.e. "and"), articles (f.e. "the") or prepositions (f.e. "in") are frequent and may not add relevant information. As example, we consider the following feature name and description: "*Eclipse Scout Project. Eclipse Scout is ~~a~~ business application framework ~~that~~ supports desktop, web ~~and~~ mobile frontends. ~~This~~ feature contains ~~the~~ Scout core runtime components.*". We apply Part-of-Speech Tagger techniques using OpenNLP [3].
- Stemming: This technique reduces the words to their root. The objective is to unify words for not to consider them as unrelated. For example "playing" will be stemmed to "play" or "tools" to "tool". Instead of keeping the

root, we keep the word with greater number of occurrences to replace the involved words. As example, in the Graphiti feature name and description we find "[...]*Graphiti supports the fast and easy creation of unified* ***graphical*** *tools, which can* ***graphically*** *display[...]*" so graphical and graphically is considered the same word as their shared stem is *graphic*. Regarding the implementation, we used the Snowball steamer [22].

### 5.2 Feature location techniques

We explain the four examples of feature location techniques. Next Section 5.3 will be dedicated to present the results of using EFLBench.

**FCA and Strict Feature-Specific (SFS) location:** FCA is used for block identification. Then, for feature location we use *Strict Feature-Specific location* that consider the following assumptions: A feature is located in a block when 1) the block always appears in the artefacts that implements this feature and 2) the block never appears in any artefact that does not implement this feature. Using this technique, the implementation of a feature is located in the plugin elements of the whole block. The principles of this feature location technique is similar to locating distinguishing features using diff sets [23]. In the Eclipse packages case, notice that, given the low number of variants and identified blocks, a lot of features will be located for the same block. In Eclipse Kepler SR2, an average of 7.25 features are located for each of the 60 blocks with a standard deviation of 13.71 features.

**FCA and SFS and Shared term:** The intuition behind this technique is first to group features and blocks with FCA and SFS and then apply a "search" of the feature words inside the elements of the block to discard elements that may be completely unrelated. For each association between feature to a block, we keep, for this feature, only the elements of the block which have at least one meaningful name shared with the feature. In other words, we keep the elements which *term frequency* (tf) between feature and element (featureElementTF) is greater than 0. For clarification, featureElementTF is defined as follows being $f$ the feature, $e$ the element and $tf$ a method that just counts the number of times that a given term appears in a given list of terms:

$$featureElementTF(f, e) = \sum_{term_i \in e.terms} tf(term_i, f.terms)$$

**FCA and SFS and Term frequency:** FCA is used for block identification and then SFS as in the previous approaches. Then, the intuition of this technique is that all the features assigned to a block competes for the block elements. The feature (or features in case of drawback) with higher *featureElementTF* will keep the elements while the other features will not consider this element as part of it.

**FCA and SFS and tf-idf:** FCA and SFS are used as in the previous approaches. The features also compete in this case for the elements of the block but a different weight is used for each word of the feature. This weight (or score) is calculated through the *term frequency - inverse document frequency* (tf-idf) value of the set of features that are competing. tf-idf is a well known technique

in IR [27]. In our context, the intuition is that words that appear more frequent through the features may not be as important as less frequent words. For example "Core", "Client" or "Documentation" are maybe more frequent across features but "CVS" or "BIRT", being less frequent, are more relevant, informative or discriminating. As in the previous approach, the feature (or features) with higher *featureElementScore* will keep the elements while the other features will not consider them. The *featureElementScore* formula is defined as follows, being $F$ the set of features that are competing for the block element.

$$featureElementScore(f, e, F) = \sum_{term_i \in e.terms} tfidf(term_i, f, F)$$
$$tfidf(term_i, f, F) = tf(term_i, f.terms) \times idf(term_i, F)$$
$$idf(term_i, F) = \log\left(\frac{|F|}{|\{f \in F : term_i \in f\}|}\right)$$

### 5.3 Results

We used the benchmark created with each of the Eclipse releases presented in Table 2. The experiments were launched using BUT4Reuse at commit ce3a002 (19 December 2015) which contains the presented feature location techniques. Detailed instructions for reproducibility are available [3]. We used a laptop Dell Latitude E6330 with a processor Intel(R) Core(TM) i7-3540M CPU@3.00GHz with 8GB RAM and Windows 7 64-bit.

After using the benchmark, we obtain the results shown in Table 3. *Precision* and *Recall* are the mean of all the features as discussed at the end of Section 4.2. The results in terms of precision are not satisfactory in the presented feature location techniques. This suggests that the case study is challenging. Also we noticed that there are no very relevant differences in the results of these techniques among the different Eclipse releases. As discussed before, given the few amount of Eclipse packages under consideration, FCA is able to distinguish

[3] `https://github.com/but4reuse/but4reuse/wiki/Benchmarks`

**Table 3.** Precision and recall of the different feature location techniques

| | **SFS** | | **SFS+ST** | | **SFS+TF** | | **SFS+TFIDF** | |
|---|---|---|---|---|---|---|---|---|
| **Release** | **Precision** | **Recall** | **Precision** | **Recall** | **Precision** | **Recall** | **Precision** | **Recall** |
| Europa Winter | 6.51 | 99.33 | 11.11 | 85.71 | 12.43 | 58.69 | 13.07 | 53.72 |
| Ganymede SR2 | 5.13 | 97.33 | 10.36 | 87.72 | 11.65 | 64.31 | 12.80 | 52.70 |
| Galileo SR2 | 7.13 | 93.39 | 10.92 | 82.01 | 11.82 | 60.50 | 12.45 | 53.51 |
| Helios SR2 | 9.70 | 91.63 | 16.04 | 80.98 | 25.97 | 63.70 | 29.46 | 58.39 |
| Indigo SR2 | 9.58 | 92.80 | 15.72 | 82.63 | 19.79 | 59.72 | 22.86 | 57.57 |
| Juno SR2 | 10.83 | 91.41 | 19.08 | 81.75 | 25.97 | 61.92 | 24.89 | 60.82 |
| Kepler SR2 | 9.53 | 91.14 | 16.51 | 83.82 | 26.38 | 62.66 | 26.86 | 57.15 |
| Luna SR2 | 7.72 | 89.82 | 13.87 | 82.72 | 22.72 | 56.67 | 23.73 | 51.31 |
| *Mean* | *8.26* | *93.35* | *14.20* | *83.41* | *19.59* | *61.02* | *20.76* | *55.64* |

blocks which may actually correspond to a high number of features. For example, all the plugins that correspond specifically to the Eclipse Modeling package, will be grouped in one block while many features are involved.

The first location technique (FCA + SFS) does not assume meaningful names given that no IR technique is used. The features are located in the elements of a whole block obtaining a high recall. Eclipse feature names and descriptions are probably written by the same community of developers that create the plugins and decide their names. In the approaches using IR techniques, the authors expected a higher increment of precision without a loss of recall but the results suggest that certain divergence exists between the vocabulary used at feature level and at implementation level.

Regarding the time performance, Table 4 shows, in milliseconds, the time spent for the different releases. *Adapt* time corresponds to the time to abstract the Eclipse packages into a set of plugin elements and get their information. The FCA time corresponds to the time for block identification. Then, the following columns show the time of the different feature location techniques. We can observe that the time performance is not a limitation of these techniques as they take around half a minute maximum.

It is out of the scope of the paper to propose innovative feature location techniques. The objective is to present the benchmark usage, show that quick feedback from feature location techniques can be obtained in the Eclipse releases case studies. In addition, we provide empirical results of four feature location techniques that can be used as baseline. Other block identification approaches can be used to further split the groups obtained by FCA as for example the clustering proposed by Salman *et al.* [25]. Other feature location techniques can make use of the available plugin and feature dependencies information as presented in Figure 2 and 3. Other works can evaluate the filtering of non-relevant domain specific words for the IR techniques (f.e. "Eclipse" or "feature") or even make use of an Eclipse domain ontology to refine feature location. Finally, meta-techniques for feature location can be proposed inspired by *ensemble learning* from the data mining research field. These meta-techniques can use multiple feature location techniques, providing better results than using each of them alone.

**Table 4.** Time performance in milliseconds for feature location

| Release | Adapt | FCA | SFS | SFS+ST | SFS+TF | SFS+TFIDF |
|---|---|---|---|---|---|---|
| Europa Winter | 2,397 | 75 | 6 | 2,581 | 2,587 | 4,363 |
| Ganymede SR2 | 7,568 | 741 | 56 | 11,861 | 11,657 | 23,253 |
| Galileo SR2 | 10,832 | 1,328 | 107 | 17,990 | 17,726 | 35,236 |
| Helios SR2 | 11,844 | 1,258 | 86 | 5,654 | 5,673 | 12,742 |
| Indigo SR2 | 12,942 | 1,684 | 100 | 8,782 | 8,397 | 16,753 |
| Juno SR2 | 16,775 | 2,757 | 197 | 7,365 | 7,496 | 14,002 |
| Kepler SR2 | 16,786 | 2,793 | 173 | 8,586 | 8,776 | 16,073 |
| Luna SR2 | 17,841 | 3,908 | 233 | 15,238 | 15,363 | 33,518 |
| *Mean* | *12,123* | *1,818* | *120* | *9,757* | *9,709* | *19,493* |

## 6 Related work

In SPL engineering several benchmarks and common test subjects have been proposed. Herrejon *et al.* proposed evaluating SPL technologies on a common artefact, a Graph Product Line [17], which variability features are familiar to any computer engineer. The same authors proposed a benchmark for combinatorial interaction testing techniques for SPLs [18]. Betty [28] is a benchmark for evaluating automated feature model analysis techniques, which has long history in software engineering research [6]. Feature location on software families is also becoming more mature with a relevant proliferation of techniques. Therefore, benchmarking frameworks to support the evolution of this field are in need.

Many different case studies have been used for evaluating feature location in software families [5]. For instance, ArgoUML variants [8] have been extensively used. However, none of the presented case studies have been proposed as a benchmark except the variants of the Linux kernel by Xing *et al.* [32]. This benchmark considers 12 variants of the Linux kernel from which a ground truth is extracted with the traceability of 2400 features to code parts. However, even if the Linux kernel can be considered as an existing benchmark, EFLBench is complementary to foster feature location research because a) it maps to a project that is plugin-based, while Linux considers C code, and b) the characteristics of the Eclipse natural language corpora is different from the Linux kernel corpora. This last point is important because it has a major influence on the IR-based feature location techniques. Finally, using the Linux kernel benchmark, the ground truth may be also constructed but there is no framework to support the experiment. EFLBench is associated with BUT4Reuse which integrates feature location techniques making easier to control and reproduce the settings of the studied techniques.

## 7 Conclusion

We have presented EFLBench, a framework and a benchmark for supporting research on feature location. The benchmark is based on the Eclipse releases and is designed to support research on software reuse in the context of software product lines. Existing and future techniques dealing with this problem can find a challenging playground that is: a) real, b) contains a valid ground-truth and c) is directly reproducible. We also demonstrated example results of four approaches using the EFLBench.

As further work we aim to create a parametrizable generator for Eclipse packages. This generator will combine different features in order to use the benchmark in special and predefined characteristics. We also aim to generalize the usage of feature location benchmarks inside BUT4Reuse providing extensibility points for other case studies. Finally, we plan to use the benchmark in order to test and report existing and innovative feature location techniques while also encouraging the research community on using it as part of their evaluation.

**Acknowledgments.** Supported by the National Research Fund Luxembourg (FNR), under the AFR grant 7898764.

## References


1. Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., Lahire, P.: Extraction and evolution of architectural variability models in plugin-based systems. Software and System Modeling 13(4), 1367–1394 (2014)
2. Al-Msie'deen, R., Seriai, A., Huchard, M., Urtado, C., Vauttier, S., Salman, H.E.: Feature location in a collection of software product variants using formal concept analysis. In: Proc. of Intern. Conf. on Soft. Reuse, ICSR 2013. pp. 302–307 (2013)
3. Apache: Opennlp (2010), `http://opennlp.apache.org`
4. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines - Concepts and Implementation. Springer (2013)
5. Assunção, W.K.G., Vergilio, S.R.: Feature location for software product line migration: a mapping study. In: Intern. Soft. Prod. Lines Conf., Companion Volume for Workshop, Tools and Demo papers, SPLC. pp. 52–59 (2014)
6. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35(6), 615–636 (2010)
7. Chen, K., Rajlich, V.: Case study of feature location using dependence graph, after 10 years. In: The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010. pp. 1–3 (2010)
8. Couto, M.V., Valente, M.T., Figueiredo, E.: Extracting software product lines: A case study using conditional compilation. In: Europ.Conf. on Software Maint. and Reeng., CSMR 2011,. pp. 191–200 (2011)
9. Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K.: An exploratory study of cloning in industrial software product lines. In: 17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013. pp. 25–34. IEEE Computer Society (2013)
10. Falleri, J.R., Dolques, X.: Erca - eclipse's relational concept analysis (2010), `https://code.google.com/p/erca/`
11. Fischer, S., Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: Enhancing clone-and-own with systematic reuse for developing software variants. In: Proc. of Intern. Conf. on Sof. Maint. and Evol (ICSME), 2014. pp. 391–400 (2014)
12. Font, J., Ballarín, M., Haugen, O., Cetina, C.: Automating the variability formalization of a model family by means of common variability language. In: SPLC. pp. 411–418 (2015)
13. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edn. (1997)
14. Grünbacher, P., Rabiser, R., Dhungana, D., Lehofer, M.: Model-based customization and deployment of eclipse-based tools: Industrial experiences. In: Intern. Conf. on Aut. Sof. Eng. (ASE). pp. 247–256 (2009)
15. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (1990)
16. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Proc. of the 30th Inter. Conf. on Soft. Eng. (ICSE). pp. 311–320 (2008)
17. Lopez-Herrejon, R.E., Batory, D.S.: A standard problem for evaluating product-line methodologies. In: Generative and Component-Based Software Engineering, Third International Conference, GCSE 2001, Erfurt, Germany, September 9-13, 2001, Proceedings. pp. 10–24 (2001)
18. Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Haslinger, E.N., Egyed, A., Alba, E.: Towards a benchmark and a comparison framework for combinatorial interaction testing of software product lines. CoRR abs/1401.5367 (2014)

19. Lopez-Herrejon, R.E., Ziadi, T., Martinez, J., Thurimella, A.K., Acher, M.: Third international workshop on reverse variability engineering (REVE 2015). In: Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015. p. 394 (2015)
20. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Automating the extraction of model-based software product lines from model variants. In: ASE 2015, Lincoln, Nebraska, USA (2015)
21. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L.: Bottom-up adoption of software product lines: a generic and extensible approach. In: Proc. of Intern. Conf. on Software Product Line, SPLC 2015. pp. 101–110 (2015)
22. Porter, M.F.: Snowball: A language for stemming algorithms. Published online (October 2001), `http://snowball.tartarus.org/`, accessed 19.11.2015
23. Rubin, J., Chechik, M.: Locating distinguishing features using diff sets. In: IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012. pp. 242–245 (2012)
24. Rubin, J., Chechik, M.: A survey of feature location techniques. In: Domain Engineering, Product Lines, Languages, and Conceptual Models, pp. 29–58 (2013)
25. Salman, H.E., Seriai, A., Dony, C.: Feature location in a collection of product variants: Combining information retrieval and hierarchical clustering. In: Intern. Conf. on Sof. Eng. and Know. Eng. SEKE. pp. 426–430 (2014)
26. Salman, H.E., Seriai, A., Dony, C.: Feature-to-code traceability in a collection of software variants: Combining formal concept analysis and information retrieval. In: Intern. Conf. on Inform. Reuse and Integr.IRI. pp. 209–216 (2013)
27. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Commun. ACM 18(11), 613–620 (1975)
28. Segura, S., Galindo, J.A., Benavides, D., Parejo, J.A., Cortés, A.R.: Betty: benchmarking and testing on the automated analysis of feature models. In: Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings. pp. 63–71 (2012)
29. Shatnawi, A., Seriai, A., Sahraoui, H.A.: Recovering architectural variability of a family of product variants. In: Software Reuse for Dynamic Systems in the Cloud and Beyond - 14th International Conference on Software Reuse, ICSR 2015, Miami, FL, USA, January 4-6, 2015. Proceedings. pp. 17–33 (2015)
30. Sim, S.E., Easterbrook, S.M., Holt, R.C.: Using benchmarking to advance research: A challenge to software engineering. In: Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA. pp. 74–83 (2003)
31. Souza, I.S., Fiaccone, R., de Oliveira, R.P., Almeida, E.S.D.: On the relationship between features granularity and non-conformities in software product lines: An exploratory study. In: 27th Brazilian Symposium on Software Engineering, SBES 2013, Brasilia, Brazil, October 1-4, 2013. pp. 147–156 (2013)
32. Xing, Z., Xue, Y., Jarzabek, S.: A large scale linux-kernel based benchmark for feature location research. In: Proced. of Intern. Conf. on Soft. Eng., ICSE. pp. 1311–1314 (2013)
33. Xue, Y., Xing, Z., Jarzabek, S.: Feature location in a collection of product variants. In: Proc. of Working Conf. on Rev. Eng., WCRE 2012. pp. 145–154 (2012)
34. Ziadi, T., Henard, C., Papadakis, M., Ziane, M., Traon, Y.L.: Towards a language-independent approach for reverse-engineering of software product lines. In: Symposium on Applied Computing, SAC 2014, 2014. pp. 1064–1071 (2014)